Name of Student: Pushkar Sane			
Roll Number: 45		**Assignment Number: 1**	
Title of Assignment: Python Assignment 1 (Data Structure & Threads in Python).			
DOP: 01-11-2023		**DOS: 03-11-2023**	
CO Mapped: **CO3, CO5**	**PO Mapped:** **PO3, PO5, PSO1, PSO2**		**Signature:**

Python Assignment No. 1

1. **Explain multithreading in Python.**

   Multithreading is a programming technique that allows a program to execute multiple threads concurrently within a single process. Threads are independent sequences of instructions that can perform tasks simultaneously. In the context of Python, the threading module facilitates the creation and management of threads.

   Here's a more detailed elaboration of multithreading:

   a. **Threads:**
      i. Threads are the smallest unit of execution within a program.
      ii. They share resources, such as memory space, within the same process.
      iii. Threads enable concurrent execution of tasks, allowing programs to perform multiple operations simultaneously.

   b. **Concurrency and Parallelism:**
      i. Concurrency: Involves the execution of multiple tasks in overlapping time periods. In Python, due to the Global Interpreter Lock (GIL), threads may execute concurrently but not necessarily in parallel across multiple CPU cores.
      ii. Parallelism: Refers to tasks running simultaneously on different CPU cores. Pure parallelism within a single Python process is limited due to the GIL.

   c. **Global Interpreter Lock (GIL):**
      i. The GIL is a mechanism in CPython (the standard Python implementation) that allows only one thread to execute Python bytecode at a time. This limitation restricts full multi-core CPU utilization, particularly for CPU-bound tasks.

   d. **Use Cases for Multithreading:**
      i. I/O-Bound Tasks: Multithreading is highly effective for I/O-bound operations, where threads can make progress while waiting for input/output tasks, such as network operations or file I/O.
      ii. Responsive Applications: Threads can be used in applications where responsiveness is crucial. Offloading certain tasks to separate threads

can prevent blocking the main execution flow, ensuring a more responsive application.

e. **Managing Threads in Python:**
   i. Python provides the threading module to create, start, and manage threads.
   ii. You can create a thread by instantiating the Thread class and specifying a function to execute as the target.
   iii. Use the start() method to start the execution of a thread.

f. **Considerations:**
   i. GIL Limitation: While multithreading is beneficial for I/O-bound tasks, the GIL restricts pure parallelism for CPU-bound operations in Python.
   ii. Thread Safety: When multiple threads access shared resources, synchronization mechanisms (such as locks) are required to prevent race conditions and ensure data integrity.

Understanding and effectively utilizing multithreading involves considering the type of tasks being performed, the limitations of the GIL, and ensuring proper synchronization and resource management among threads to achieve efficient and safe concurrent execution in Python.

2. **Define the two types of data structures and give differences between them.**

Data structures in computer science are categorized into two primary types: arrays and linked lists.

1. **Arrays:**
   a. Definition: An array is a sequential collection of elements, where each element is identified by an index or a key. Elements are stored in contiguous memory locations.
   b. **Characteristics:**
      - Random Access: Elements can be accessed directly by their index.
      - Fixed Size: Arrays have a fixed size defined during initialization, making resizing complex and, often, requiring the creation of a new array.
      - Homogeneous Elements: Arrays typically store elements of the same data type.

- Memory Allocation: Memory for an array is allocated in a contiguous block.

c. **Advantages:**

- Efficient random access to elements.
- Direct memory addressing allows for faster access.
- Ideal for scenarios where the size of the collection is known in advance.

d. **Disadvantages:**

- Fixed size can lead to memory wastage or insufficient space.
- Insertions and deletions may be inefficient as they might require shifting elements to accommodate changes.

2. **Linked Lists:**

a. **Definition:** A linked list is a linear collection of elements where each element is a separate object called a node. Nodes in a linked list are connected through pointers or references.

b. **Characteristics:**

- Dynamic Size: Linked lists can grow or shrink dynamically as nodes are added or removed.
- No Fixed Memory Requirement: Nodes in a linked list are not necessarily stored in contiguous memory.
- Various Types: Different types of linked lists exist—singly linked, doubly linked, and circular linked lists—based on how nodes are connected.

c. **Advantages:**

- Dynamic size allows for efficient memory usage.
- Ease of insertion and deletion operations, as it involves just changing pointers.

d. **Disadvantages:**

- Inefficient random access; to access an element, the list must be traversed from the beginning.
- Higher memory overhead due to the need for storing references or pointers to connect nodes.

3. **Differences:**
    a. Memory Allocation:
        - Arrays allocate memory in a contiguous block, while linked lists use non-contiguous memory by connecting nodes through pointers.
    b. Random Access:
        - Arrays offer direct/random access to elements via indexing, whereas linked lists require traversal from the beginning to access a specific element.
    c. Size Flexibility:
        - Arrays have a fixed size defined during initialization, while linked lists can grow or shrink dynamically.
    d. Insertion and Deletion:
        - Insertions and deletions in arrays can be less efficient as they may require shifting elements. Linked lists offer more efficient insertion and deletion as they involve changing pointers.
    e. Memory Efficiency:
        - Arrays might waste memory if not fully utilized due to their fixed size. Linked lists are more memory-efficient as they use memory only as needed.

Both arrays and linked lists have their strengths and weaknesses. The choice between them depends on the specific requirements of the application or problem at hand. Arrays are suitable for scenarios requiring random access and a fixed-size collection, while linked lists are preferred for dynamic collections with frequent insertions and deletions.

3. **Explain the concept of stack and Queue.**

Stack: A stack is a linear data structure that follows the Last-In, First-Out (LIFO) principle. It works much like a stack of plates where you can only remove the top plate (the last one placed) before accessing the rest.

Essential operations of a stack are:

a. **Push:** Adding an element to the top of the stack.
b. **Pop:** Removing the top element from the stack.
c. **Peek or Top:** Viewing the top element without removing it.
d. **Empty:** Checking if the stack is empty.

e. **Size:** Determining the number of elements in the stack.

Key characteristics of a stack:

a. Elements are added and removed from one end, often referred to as the "top" of the stack.
b. It operates on the principle of Last-In, First-Out (LIFO).
c. Common implementations are via arrays or linked lists.

Queue: A queue is a linear data structure that follows the First-In, First-Out (FIFO) principle. It's similar to people waiting in a line (queue) – the first person to join is the first to leave.

Essential operations of a queue are:

a. **Enqueue:** Adding an element to the back or rear of the queue.
b. **Dequeue:** Removing an element from the front or head of the queue.
c. **Front:** Viewing the front element without removing it.
d. **Empty:** Checking if the queue is empty.
e. **Size:** Determining the number of elements in the queue.

Key characteristics of a queue:

a. Elements are added to the rear and removed from the front of the queue.
b. It operates on the principle of First-In, First-Out (FIFO).
c. Common implementations are via arrays or linked lists.

Key Differences:

1. **Principle:**
    a. Stack follows Last-In, First-Out (LIFO).
    b. Queue follows First-In, First-Out (FIFO).
2. **Insertion and Deletion:**
    a. In a stack, elements are inserted and deleted from the top.
    b. In a queue, elements are inserted at the rear and deleted from the front.
3. **Usage:**
    a. Stacks are used in function calls, expression evaluation (e.g., postfix notation), undo mechanisms.
    b. Queues are used in scheduling, breadth-first search, printing tasks, etc.

4. **Operations:**
    a. Common operations for both include checking if empty and determining the size.
    b. Stacks have "push" and "pop" operations, while queues have "enqueue" and "dequeue."

Both stacks and queues have their unique properties that make them suitable for different applications based on their behavior (LIFO for stacks and FIFO for queues). Understanding their principles and operations is essential for efficient usage in various algorithms and applications.

4. Write an algorithm to implement insertion operation of the queue.

Insertion (Enqueue) Operation in a Queue:

Algorithm:

a. Input: Queue Q, element to be inserted (new_item).
b. Check if the queue is full (if there's a maximum size limit):
    - If the queue is full, return an error or resize the queue if possible.
c. Add the new element (new_item) to the end (rear) of the queue:
    - Append new_item to the end of the data structure representing the queue.
d. Update the rear pointer (if using a linked list implementation) or maintain the rear index/variable (if using an array-based implementation).

Explanation:

1. **Input:** The algorithm takes the queue data structure (represented by an array, linked list, or any appropriate structure) and the new element to be inserted (new_item).
2. **Queue Full Check:** If the queue has a maximum size limit and it's reached, the algorithm should handle the queue being full, either by returning an error or by implementing a resizing strategy to accommodate more elements.
3. **Insertion at Rear:** The new element (new_item) is added at the end (rear) of the queue. If using an array-based implementation, you would typically append the new element to the end of the array. For a linked list, you'd create a new node with the data and connect it to the last node, updating the pointers accordingly.

4. **Rear Pointer/Update:** In a linked list-based queue, you'd update the "rear" pointer to point to the newly added node. In an array-based queue, you'd maintain a rear index/variable to track the end of the queue after insertion.

The insertion operation in a queue (enqueue) involves adding elements to the rear or end of the queue, maintaining the queue's integrity in terms of order (FIFO - First-In, First-Out). The algorithm ensures that new elements are properly added and accounted for in the queue data structure.